Science of Computer Programming 5 (1985) 201-218
North-Holland

A METHOD AND A LANGUAGE FOR CONSTRUCTING ITERATIVE PROGRAMS

Jean-Pierre FINANCE and Jeanine SOUQUIERES
Centre de recherche en Informatique de Nancy, 54506 Vandoeuvre-les-Nancy Cedex, France

Communicated by M. Sintzoff
Received July 1983
Revised September 1984

Abstract. This paper proposes a programming method whose main idea is to give a simple static solution to a problem, taking execution details into account afterwards. The method proposes the following approach to the user: first, the result is defined concisely using a definition scheme, which introduces a number of intermediate objects. These intermediate objects characterise subproblems which have to be solved. This process is repeated recursively on these intermediate objects, until finally the data of the problem are introduced. The support language for the method is static and uses 'sequences' as a basic data type.

In a second paper we tackle the problem which Jackson terms 'structure clashes'. The same method is used, taking account of efficiency considerations. Such considerations lead to the introduction of program transformation rules.

Introduction

Important features of the method reflected in the language are structured and static definitions, modularity, and the use of sequence as a basic data type. The construction of an algorithm is top-down. Objects are known by two definitions: an informal one which is used for documentation purposes and a formal one.

One of the difficulties in programming lies in reconciling the notion of efficiency with the notions of clarity, good organization, modifiability, reliability and legibility. In order to avoid this difficulty, we divide the programming task into two major stages:

- first, we use a method for deriving a program from the specification of the problem concerned, as easily and systematically as possible; the program will be correct but perhaps inefficient.
- second, a more efficient program is produced by applying transformations, which preserve its meaning.

This idea is developed here in the particular framework of the programming method MEDEE, designed at Nancy (cf. Pair [13], Bellegarde et al. [2]). We are especially concerned with multi-level sequences or problems of 'structure clashes'. In such problems, there is no direct relationship between the input and output data

types, which are 'incompatible'. The method is founded upon the introduction of intermediate sequences, well-suited to the definition of the output result.

The problem we have to solve can be subdivided into three independent problems (Souquières [17]):
- the choice of intermediate structures well-suited to the construction of the desired result;
- the definition of the result using these intermediate objects: the algorithm for defining the result is developed by way of suitably chosen data structures; the input data structures are not taken into account at this stage;
- definition of the intermediate objects in terms of the actual input data structures.

For a wide class of problem, such as text processing, business data processing, sequential file processing, the objects manipulated are sequences. The intermediate objects are also sequences. This kind of algorithm is the subject of a particular study developed in a second paper [18].

Once the algorithm has been constructed, two attitudes can be adopted towards the intermediate sequences, depending on the specific problem and on context constraints:
- they may remain in the final solution,
- they are eliminated, whenever possible.

Our objective here is to give tools for implementing the second approach: the final version of the algorithm can be understood as the end result of a process of successive transformations. This second approach is chosen for reasons of efficiency. We remark, however, that this choice makes later phases of the algorithm difficult to modify.

This paper and the companion paper [18] attempt to investigate several points:
- a program construction method,
- an associated language,
- the definition of transformation techniques,
- the definition of a formal framework so as to justify the transformations and to describe more precisely the language and the tools developed,
- the provision of a 'help' system for the user of this method.

The two papers are organised as follows:
- in this one, we present the method and the language, in both an intuitive and a formal way. A characteristic of this work is the description of the syntax and semantics of the language using algebraic data types.
- the second one [18] presents an example of the manipulation of intermediate sequences. We introduce a set of transformations rules with a semantic justification for each particular rule. We then apply these transformations to the example.

The method and the language

We first give an informal presentation of the method for program construction using a toy example. In Section 2 we introduce the syntax of the language together

with an informal description of the semantics of its main constructs. In Section 3 we describe formally an algebraic semantics of the language. Finally, we compare our approach with other concepts used in program construction. We also refer to the application of this method in teaching. Then we propose an extension of this method to solve problems of 'structure clashes'.

1. Informal presentation of the method for program construction

Example. Let us find, for a given n, the approximate values of $sin(k * \pi/n)$ for k in the range $[0 \ldots n]$, where the relative error, *epsilon*, is given.

We use the formula

$$sin(x) = \sum_{i=0}^{n} \frac{(-1)^i x^{2*i+1}}{(2*i+1)!}$$

with a relative error less than the last term of the sum. How can we construct the corresponding algorithm?

The result consists of $n+1$ lines of printed approximate values, each of them being defined in a similar manner. Hence we use an iterative definition to construct the result:

lines = **iter** * *sin* **using** *k* **in** $1 \rightarrow n$

where n denotes an input datum and k denotes the iteration index.

"$*sin$" is a module, or set of definitions, which defines a value to be printed.

Usually, a definition introduces intermediate results or input data. Their names are systematically written in a glossary, together with the text of an informal specification. As is the case here, a definition can also introduce modules such as "$*sin$": their names and informal specifications are also written in the glossary. Thus the beginning of the algorithm appears in the form:

- *lines* **text** sequence of printed lines $*sin$ **module** specifies the printing of an approximate value of *sinus* - *k* **sequence** of integer	*lines* = **iter** $*sin$ **using** *k* **in** $1 \rightarrow n$

The minus sign (−) in front of a line in the left part means that the name has been explicitly defined in the right part.

Now, we have to define n:

n = **input**

This definition does not introduce any new intermediate object. The definition of the main module is complete and it remains to define the module "**sin*".

Let *line* be an element of *lines*. This result denotes the printing of an approximate value of *sinus*. So:

line = **write** *sinus*

where "**write**" is a primitive of the language, "*sinus*" is the approximate value of $sin(k * \pi/n)$ computed from the formula $\sum_{i=0}^{n}(-1)^i x^{2*i+1}/(2*i+1)!$. So, we can define *sinus* as the last term of the sequence, defined by the iteration:

sinus = **last until** *finished* **iter** **approach*

"*finished*" is a sequence of booleans whose value becomes true when the required precision is obtained, "**approach*" is the module defining the current approximation to *sinus*, and the current value of *finished*.

The module **sin* is compound of

**sin*	
- *sinus* **real** last term of the sequence of approximate values of $sin(k * \pi/n)$ **approach* **module** defines $sinus_p$ and $finished_p$	*line* = **write** *sinus* sinus = **last until** *finished* **iter** **approach*

The next step consists in defining the module "**approach*", and, more precisely, the two results $finished_p$ and $sinus_p$.

$sinus_p$ is recursively defined in terms of $sinus_{p-1}$ as:

$sinus = @sinus + t.$

Remark. The symbol @ is used for the previous value of an iterated object (*@sinus* stands for $sinus_{p-1}$).

t is a real and denotes the term:

$$\frac{(-1)^p x^{2*p+1}}{(2*p+1)!}$$

an explicit definition of t is therefore

$$t = \frac{-@t * x * x}{(2 * p+1) * (2 * p)}$$

**approach*	
- t **real** $\dfrac{(-1)^p x^{2*p+1}}{(2*p+1)!}$ - x **real** $k\pi/n$ - p **integer** *current index*	$sinus = @sinus + t$ $t = \dfrac{-@t * x * x}{(2*p+1)*(2*p)}$ $p = @p + 1$ $x = k * \pi/n$

"*finished*" is true when the absolute value of the last computed term of the series is less than *epsilon*, so:

- *epsilon* **real** precision	*finished* $= (abs(t) < epsilon)$ *epsilon* = **input.**

Is the algorithm completely designed? We use iterated variables, t, p. They have to be initialised. Therefore the instructions:

$$p_0 = 0, \qquad t_0 = 0$$

should be added to the module **sin.*

Remark. In the module **approach*, x and *epsilon* are constant sequences. They can be defined once in the module **sin.*

To obtain an algorithm, we put the definitions introduced together in the same table. This table is in two separate parts: the left side, or the lexicon, contains informal definitions while the right side contains explicit definitions. The presentation is modular.

- *lines* **text** sequence of printed lines - **sin* **module** defines the printing of an approximate value of *sinus* - n **integer** input data - k **sequence** of integer	*lines* = **iter** **sin* **using** k **in** $1 \to n$ n = **input**
**sin*	
- *sinus* **real** sequence of the approximate values of $sin(k\pi/n)$ computed from a formula - *finished* **boolean** sequence of stop conditions	*line* = **write** *sinus* *sinus* = **last until** *finished* **iter** * *approach* $sinus_0 = x$ $x = k * \pi/n$ *epsilon* = **input**

- $*approach$ **module** defines $sinus_p$ and $finished_p$ recursively in terms of $sinus_{p-1}$	$p_0 = 0$ $t_0 = 0$ $finished_0 = (abs(t) < epsilon)$
$*approach$	
- t **real** $\dfrac{(-1)^p x^{2*p+1}}{(2*p+1)!}$ - x **real** $k\pi/n$ - p **integer** current index - $epsilon$ **real** expected precision	$sinus = @sinus + t$ $t = \dfrac{-@t * x * x}{(2*p+1)*2*p}$ $p = @p + 1$ $finished = (abs(t) < epsilon)$

We can summarise our approach as follows: first we try to define the result using definitions and primitives of the language and intermediate results. Every intermediate object is then written in the glossary with an informal definition. The next step consists in repeating the preceding one for one of the identifiers of the glossary not yet explicitly defined. This process is repeated for every identifier of the glossary until they are all defined.

2. Language presentation

The language developed here is a definitional language adapted to the specification of recursive algorithms and oriented towards program construction. The differences between the language and classical programming languages lies in its static and methodological features. Rather than describing the computation steps, it allows us to express the relation between data and results. It also allows a gradual solution of a problem.

Like in LUCID (Aschroft [1]), the fundamental object in this language is the *sequence*: it is generally defined by iteration. Whereas the goal in LUCID is to probe programs, in MEDEE it is the intention that programs be constructed. We can compare our approach to that of Reddy [16].

Every MEDEE program is composed of a set of modules, each module being itself a set of definitions. Sequences are basic objects of the language and are often defined by iteration.

2.1. Basic constructions of the language

We use one of the three definitional forms:

- *simple definition*: used when an object can be defined by an algebraic expression.

We denote this by

$$x = exp(v, \ldots z)$$

where $v, \ldots, z$ are intermediate objects.

- *conditional definition* such as

$$x = \textbf{if } condition \textbf{ then } * cx1 \textbf{ else } * cx2$$

where *condition* is a boolean, $*cx1$ denotes the module (or set of definitions) defining x when '*condition*' is true, $*cx2$ denotes the module defining x when '*condition*' is false.

- *iterative definition* used to define the sequences. Sequences can be expressed in three different forms:

(i) iterative definitions using a sequence S:

$$U = \textbf{iter } *cu \textbf{ using } S.$$

This defines the sequence U. The current term u_i of U is defined as a function of s_i, the current term of S. U and S are bijective. '*cu' denotes the module or set of definitions which associates the term u_i in U with each s_i in S, S being an ordinary sequence.

This form of iteration is very easy to use because at the time we define the sequence U, we are not concerned with the way the terms of the sequence S are enumerated.

Remark. When S is an integer interval, we use the form:

$$U = \textbf{iter } *cu \textbf{ using } k \textbf{ in } a \rightarrow b$$

k is an index varying in the interval $a \rightarrow b$, that is from a to b in steps of 1.

(ii) iterative definitions using a sequence S and a stop condition:

$$U = \textbf{until } stop \textbf{ iter } *cu \textbf{ using } S$$

This form of definition is used for a sequence whose initial values are not precisely known. The sequence U depends on the sequence S and on a condition (*stop*). This condition is defined in terms of the current elements of S and U.

(iii) iterative definitions without an explicit domain of definition:

$$U = \textbf{until } stop \textbf{ iter } *cu \textbf{ with } S.$$

Unlike the previous cases, we do not know how the sequence S is related to U at the time this definition is written. There is no bijective relation between s_i and u_i, but rather some more complex relation.

2.2. Syntax

The abstract syntax of the language is described using a Backus-like notation; a more exhaustive description is given in Souquières [17]:

specification::=(*module*)*
module::=*identmod*, (*definition*)*
definition::=*informaldef*, *formaldef*
informaldef::=(*ident*, *typident*, *comment*)*
formaldef::=*s-expr* | *c-expr* | *i-expr*
s-expr::=*expr* | *identmod*
c-expr::=*cond*, *s-expr*, *s-expr*
i-expr::=[*cond*,]*s-expr*[, *domaine*].

Remark. A definition is composed of two parts:
- informal definition (defined by *informaldef*) and specified in natural language,
- formal definition (*formaldef*) which is a constructive definition in the proposed language.

3. Algebraic semantics of the language

Other semantics concerning MEDEE have been given by Finance [6] and Finance and Lescanne [7].

3.1. Basic ideas

The semantics of a program is a function whose domain is the set of input values and whose range is the set of output values. In order to define this function, we use the algebraic data type framework. We may summarize our approach using the following schema:

abstract syntax	→ abstract data type objects representing language sentences	→ functions associated with the language statements

We need to introduce:
- an abstract syntax of the language,
- abstract data type objects to represent the statements of the language,
- functions associated with the language statements: they must introduce a type (FUNC) for functions of a tuple of values (a value is either a simple value or a sequence).

This approach may be compared with those of Pair [14, 15], Gaudel [8], Broy and Wirsing [4, 5]. Our approach is simpler since we do not introduce the notion of variable, as it is usually understood in imperative languages, and of state.

3.2. The sequence data type

We use a type sequence named SEQ as a primitive type in MEDEE. SEQ is defined as a parametrized data type over the primitive data type v.

We first introduce the parameterized data type PAIR[V] with two constructors:

pnil		→ PAIR	
tuple	: V * V	→ PAIR	
firstp	: PAIR	→ V	gives the first element of the PAIR
secondp	: PAIR	→ V	gives the second element of the PAIR

3.2.1. Sequence data type operators

The objects of sort SEQ are functions constructed from the following primitives operators:

snil	:	→ SEQ[V]	
cons	: SEQ[V] * VAL	→ SEQ[V]	
first	: SEQ[V]	→ VAL	
last	: SEQ[V]	→ VAL	
head	: SEQ[V]	→ SEQ[V]	
tail	: SEQ[V]	→ SEQ[V]	
until	: SEQ[V] * BOOLEAN	→ SEQ[V]	
lg	: SEQ[V]	→ INTEGER	
concat	: SEQ[V] * SEQ[V]	→ SEQ[V]	
issubseq	: SEQ[V] * SEQ[V]	→ BOOLEAN	
dec	: SEQ[PAIR[V]]	→ PAIR[SEQ[V]]	gives a pair of sequences from a sequence of pair of values.

cons and *snil* are the constructors of the data type SEQ. An induction principle for sequences can be written as follows:

> for every property φ upon SEQ such that $\varphi(snil)$ is true and $\forall$ V $\in$ VAL, $\forall S \in$ SEQ[V], $\varphi(S) \Rightarrow \varphi(cons(S, v))$ **then** $\varphi(s)$ *true* for every $S \in$ SEQ[V]

3.2.2. Axioms

To preserve the correctness of the definition of some operators, we need to introduce an extension of the type VAL for the undefined element. Let Vl be an extension of VAL so that

Ω : → Vl

(a) $first(snil) = \Omega$

$first(cons(s, v)) =$ **if** $s = snil$ **then** v **else** $first(s)$

(b) $last(snil) = \Omega$

$last(cons(s, v)) = v$

(c) $head(snil) = snil$

$head(cons(s, v)) = s$

(d) $tail(snil) = snil$
$tail(cons(s, v)) = \textbf{if} = snil \textbf{ then } snil$
$\textbf{else } cons(tail(s), v)$

(e) $until(snil, stop) = snil$
$until(cons(s, v), stop) = \textbf{if } stop = true \textbf{ then } s$
$\textbf{else } cons(s, v)$

(f) $lg(snil) = 0$
$lg(cons(s, v)) = lg(s) + 1$

(g) $concat(s1, snil) = s1$
$concat(s1, cons(s2, v)) = cons(concat(s1, s2), v)$

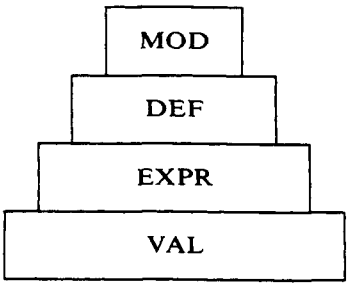
(h) $issubseq(snil, s) = snil$
$issubseq(cons(s1, v), s2) =$
$\textbf{if } first(s1) = first(s2) \textbf{ then } issubseq(cons(tail(s1), v), tail(s2))$
$\textbf{else } issubseq(cons(tail(s1), v), s2).$

(i) Applied to a sequence of pairs of elements, *dec* returns a pair of sequences of elements. *dec* is the inverse operator to the merging of two sequences.

$firstp(dec(s12)) =$
$\textbf{if } s12 = snil \textbf{ then } snil$
$\textbf{else } cons(firstp(first(s12)), firstp(dec(tail(s12))))$
$secondp(dec(s12)) =$
$\textbf{if } s12 = snil \textbf{ then } snil$
$\textbf{else } cons(secondp(first(s12)), secondp(dec(tail(s12)))).$

3.3. *The abstract syntax*

An algebraic description of all the language constructions has been designed; for more details see [17]. To present the sentences of the language, we need to introduce the following data types:



for Modules
for Definitions
for Expressions
for Values

To the data type VAL, we associate an evaluation function and to the data type EXPR a substitution function.

Let us examine the data type *Module* (abbreviated to MOD) by way of an example. A module is a set of definitions associated with one or more results. It may be considered as the union of three sets of definitions:
- in the first set, the initial values of the recursive objects are defined (this set may be empty),

- the second set consists of the result definitions,
- the third set comprises the intermediate result definitions.

The data type MOD depends on the data types:

DEF
BOOLEAN
SIDENT (for sequence of identifiers, SIDENT : ID * ID · · · * ID).

3.3.1. *Module data type operators*

modnil	:	→ MOD	
addinit	: MOD * DEF	→ MOD	add an initialization definition
addres	: MOD * DEF	→ MOD	add a result definition
addrec	: MOD * DEF	→ MOD	add an intermediate definition
result	: MOD	→ SIDENT	set of the result identifiers
mrightid	: MOD	→ SIDENT	set of the right-hand side identifiers of the module
init	: MOD	→ MOD	set of initialization definitions
rec	: MOD	→ MOD	set of result definitions
subtract	: MOD * DEF	→ MOD	subtract a definition of a module
modmerge	: MOD * MOD	→ MOD	merging of two modules
isdefined	: MOD * SIDENT	→ BOOLEAN	is the identifier defined in the module?
isin	: MOD * DEF	→ BOOLEAN	does the definition belong in the module?
isused	: MOD * SIDENT	→ BOOLEAN	is the identifier used in the module?
idright	: MOD * DEF	→ SIDENT	set of right-hand side identifiers of the module introduced by the definition
depend	: MOD * SIDENT * SIDENT	→ SIDENT	to express non-circularity

(if the object x is defined in terms of the object y, then the definition of the object y cannot contain occurrence of x).

modnil, *addinit*, *addres* and *addrec* are the constructors of the data type MOD. An induction principle for the type module can be written as follows:

For every property φ over MOD
if $\varphi(modnil)$ is *true* and **if** for every $m \in$ MOD, for every $d \in$ DEF
$\varphi(m)$ is *true* **implies** $\varphi(addres(m, d))$ **and** $\varphi(addrec(m, d))$ **and**
$\varphi(addinit(m, d))$ are *true* **then** $\varphi(m)$ is *true* for every module m.

We choose an initial algebra as the formalism for algebraic specification. With every sentence of the language, we associate a term of the abstract data type, its congruence class being given by the axioms.

3.3.2. Preconditions

prec $addinit(m, d) \equiv isin(m, d) = false$
prec $addres(m, d) \equiv isin(m, d) = false$
prec $addrec(m, d) \equiv isin(m, d) = false$
prec $subtract(m, d) \equiv isin(m, d) = true.$

We can merge two modules $m1$ and $m2$ only if they define different object.

prec $modmerge(m1, m2)$
$\equiv \forall x, \forall y,$
$(isdefined(m1, x) \Rightarrow$ **not** $isdefined(m2, x))$
and $(isdefined(m2, y) \Rightarrow$ **not** $isdefined(m1, y)).$

3.3.3. Axioms

$\forall d, d1, d2 \in$ DEF
$\forall m, m1, m2 \in$ MOD.

To be more precise, it requires to introduce operations on the sets of identifiers.

(a) $result(modnil) = \omega$
$result(addinit(m, d)) = result(m)$
$result(addres(m, d)) = result(m) \cup leftpart(d)$
$result(addrec(m, d)) = result(m).$

leftpart is an operator upon DEF giving the names of the results of d. ω denotes the undefined element of the data type MOD and $\perp$ the undefined element of SIDENT.

(b) $mrightid(modnil) = \perp$
$mrightid(addinit(m, d)) = mrightid(m) \cup idright(m, d)$
$mrightid(addres(m, d)) = mrightid(m) \cup idright(m, d)$
$mrightid(addrec(m, d)) = mrightid(m) \cup idright(m, d).$

idright gives the names of identifiers used in the right-hand side of a definition in a given module.

(c) $init(modnil) = \omega$
$init(addinit(m, d)) = addinit(init(m), d)$
$init(addrec(m, d)) = init(m)$
$init(addres(m, d)) = init(m)$

(d) $depend\ (modnil, x, y) = false$
$depend(addinit(m, d), x, y) = x \in leftpart(d)$
and $y \in idright(m, d)$ **or** $depend(m, x, y)$
$depend(addrec(m, d), x, y) = x \in leftpart(d)$ **and** $y \in idright(m, d)$
or $y \in leftpart(d)$ **and** $@x \in idright(m, d)$ **or** $depend(m, x, y)$
$depend(addres(m, d), x, y) = x \in leftpart(d)$ **and** $y \in idright(m, d)$
or $y \in leftpart(d)$ **and** $@x \in idright(m, d)$ **or** $depend(m, x, y)$

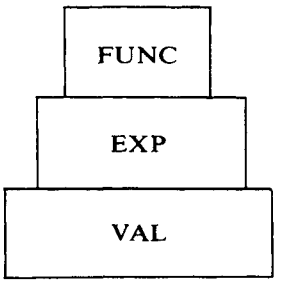
(e) $rec(modnil) = \omega$
$rec(addinit(m, d)) = rec(m)$
$rec(addrec(m, d)) = addrec(rec(m), d)$
$rec(addres(m, d) = addres(rec(m), d)$.

An algebraic description of the sentences of the language enables us to define formally the transformations over the language.

3.4. *The semantics function*

We associate a semantics function with each language construct. The goal of these functions is to enable the transformation rules to be justified.

To define these functions, we need the data types:



for Functions

for Expressions

for Values

The terms of sort FUNC are functions constructed from the following primitives functions:

λ	: SIDENT * EXP	→ FUNC	
cond	: BOOLEAN * FUNC * FUNC	→ FUNC	regular conditional operator
comp	: IDENT * FUNC * FUNC	→ FUNC	composition of functions
par	: FUNC * FUNC	→ FUNC	definition of several results
decf	: FUNC	→ FUNC * FUNC	split one result into two
arity	: FUNC	→ INTEGER	gives the number of result parameters
it	: MOD * SEQ * SEQ	→ FUNC	semantics function associated with **iter** * *cu* **using** *S*
itu	: MOD * BOOLEAN * SEQ * SEQ	→ FUNC	semantics function associated with **until** *stop* iter * *cu* **using** *S*
itw	MOD * BOOLEAN * SEQ * SEQ	→ FUNC	semantics function associated with **until** *stop* **iter** * *cu* **with** *S*

Some preconditions must hold.

For every $f, g \in$ FUNC, $x, y \in$ IDENT, $exp \in$ EXP:

$$\begin{aligned}
&\textbf{prec } cond(c, f, g) &&\equiv arity(f) = arity(g)\\
&\textbf{prec } comp(x_i, f, g) &&\equiv arity(g) = 1\\
&\textbf{prec } par(f, g) &&\equiv arity(par(f, g)) \neq 1\\
&\textbf{prec } decf(f) &&\equiv arity(f) \neq 1.
\end{aligned}$$

By definition $arity(exp) = 1$.

Remark. *arity* is an operator applied on the domain of a function.

By definition:

(a) $\lambda.x1 \ldots xn.exp = f \Rightarrow f(u1, \ldots, un) = eval(exp(x1/u1, \ldots, xn/un)$
eval is an evaluation operator and / is a substitution operator.

(b) $cond(true, f, g) = f$
$cond(false, f, g) = g$

(c) $comp(x_i, f, g) = \lambda x_1 x_2 \ldots x_{i-1} y_1 \ldots y_p x_{i+1} \ldots x_n$
$f(x_1, \ldots, x_{i-1}, g(y_1, \ldots, y_p), x_{i-1}, \ldots, x_n).$

3.4.1. Semantics of a definition

We express the semantics of a definition d belonging to the module m by

$$Sem[\![d]\!] = \lambda.\ idright(m, d).\ rightpart(d)$$

where *rightpart* denotes the formal definition, $isin(m, d)$ is *true.*

Remark. To express the semantics of the proposed language, we sometimes start with the abstract syntax of the language and sometimes with the abstract data type *Module.* In fact, in our approach, we consider the syntax of definitions and expressions as a basic abstract data type.

(a) simple definition:

$$Sem[\![x = exp]\!] = \lambda.\ idright(m, x = exp).\ exp$$

(b) conditional definition:

$$Sem[\![x = \textbf{if } c \textbf{ then } *cx1 \textbf{ else } *cx2]\!] = cond(c, Sem[\![*cx1]\!], Sem[\![*cx2]\!])$$

The semantics of a module, denoted by $Sem[\![*cx1]\!]$ will be defined in the next paragraph.

(c) iterative definitions:

(c.1) Form: $u = \textbf{iter } *cu \textbf{ using } l$

$$Sem[\![u = \textbf{iter } *cu \textbf{ using } l]\!] = \textbf{it } (*cu)(u_0, l)$$

it denotes the semantics function of the iteration. It is a recursive function depending on the module $*cu$. The first call of this function implies the initialization of the sequence u, and the sequence l.

Definitions

$$u_0 = Sem[\![init(*cu)]\!]$$
$$\mathbf{it}(*cu)(u, ll) = cond(ll = snil,\ u,\ concat(u, \mathbf{it}(*cu)(u', tail(ll))))$$
$$u' = Sem[\![rec(*cu)]\!](u, first(ll))$$

(c.2) Form: $u = \mathbf{until}\ stop\ \mathbf{iter} * cu\ \mathbf{using}\ l$

$$Sem[\![u = \mathbf{until}\ stop\ \mathbf{iter} * cu\ \mathbf{using}\ 1]\!] = itu(*cu)(u_0, stop_0, l)$$

where

$$u_0, stop_0 = Sem[\![init(*cu)]\!]$$
$$\mathbf{itu}(*cu)(u, stop, ll) = cond(stop\ \mathbf{or}\ ll = snil,\ u,\ concat(u, \mathbf{it}(*cu)(u', tail(ll))))$$
$$u', stop = Sem[\![rec(*cu)]\!](u, first(ll)).$$

(c.3) Form: $u = \mathbf{until}\ stop\ \mathbf{iter} * cu\ \mathbf{with}\ l$

$$Sem[\![u = \mathbf{until}\ stop\ \mathbf{iter} *cu\ \mathbf{with}\ l]\!] = \mathbf{itw}(*cu)(u_0, stop_0, l)$$

where

$$u_0, stop_0, l_0 = Sem[\![init(*cu)]\!]$$
$$\mathbf{itw}(*cu)(u, stop, ll) = cond(stop, u, concat(u, \mathbf{it}(*cu)(u', ll'')))$$
$$u', stop, ll'' = Sem[\![rec(*cu)]\!](u, ll')$$
$$ll = concat(ll', ll'').$$

In this form, the sequence l is defined in the module $*cu$; l is an intermediate sequence of $*cu$.

3.4.2. Semantics of a module

A module is an ordered set of definitions. This ordering is not the execution ordering, it is what is may be termed a deductive order: the algorithm constructed starting with the definition of the result.

Definitions.

(1) $Sem[\![modnil]\!] = \omega$

(2) $Sem[\![addinit(m, d)]\!] = \mathbf{if}\ isused(m, leftpart(d))$
$\mathbf{then}\ par(comp(leftpart(d), Sem[\![m]\!], Sem[\![d]\!]), Sem[\![d]\!])$
$\mathbf{else}\ par(Sem[\![m]\!], Sem[\![d]\!])$

(3) $Sem[\![addres(m, d)]\!] = par(Sem[\![m]\!], Sem[\![d]\!])$

(4) $Sem[\![addrec(m, d)]\!] = comp(leftpart(d), Sem[\![m]\!], Sem[\![d]\!])$

(5) $Sem[\![modmerge(m1, m2)]\!] = par(Sem[\![m1]\!], Sem[\![m2]\!])$

(6) Property of the operator *comp* in relation to the semantics of a module:

$$comp(x, Sem[\![m1]\!], Sem[\![m2]\!]) = Sem[\![modmerge(m1, m2)]\!]$$

with the precondition: $isused(m1, x) = true$ **and** $x = result(m2)$

Demonstration of property (6): Let

$$S1 = comp(x, Sem[\![m1]\!], Sem[\![m2]\!])$$
$$= \lambda.idm1.idm2.Sem[\![m1]\!](idm1, Sem[\![m2]\!](idm2), idm2)$$

using the definition of the *comp* operator.

As $result(m1) \neq result(m2)$, we can demonstrate by induction on definition (3) that

$$Sem[\![m1]\!](idm1, Sem[\![m2(idm2)]\!], idm2)$$
$$= par(Sem[\![m1]\!](idm1), Sem[\![m2]\!](idm2)).$$

So

$$S1 = \lambda.idm1.idm2.par(Sem[\![m1]\!](idm1), Sem[\![m2]\!](idm2))$$
$$= \lambda.idm1.idm2.Sem[\![modmerge(m1, m2)]\!](idm1, idm2)$$

by virtue of definition (5):

$$= Sem[\![modmerge(m1, m2)]\!].$$

3.4.3. Semantics of a program

A program A is composed of a basic module, BM, using some secondary modules. These modules use themselves submodules in a hierarchical decomposition

$$Sem[\![A]\!] = Sem[\![BM]\!].$$

4. Conclusion

We have presented here a method and an associated programming language, MEDEE, to solve problems in an iterative way. The method is top-down and the main idea consists in solving a problem statically and explicitly before taking into account execution details. It differs from the notion of structured programming in that in order to define a problem we refer to the results to be obtained rather than the actions needed to compute them. We offer the user a tight guideline to solve his problem. In summary, the method is to define the desired result by using a primitive of the language and introducing some necessary intermediate objects. Then iterate this process on the intermediate objects until they are all defined, i.e. until the data of the problem are introduced.

Some of the semantics of the language have been shown [7]. In the first place they aim at giving a mathematical meaning to a MEDEE specification. They further allow the definition of proof rules [6] and the justification of algorithm transformations (see [18, Section 3]).

We are at present obtaining considerable experience in the use of the system. In particular the method has been used for education at the University of Nancy, for further education of teachers in secondary schools, and for master of computer science applied to management. An evaluation of the use of the method for teaching programming to beginners has been done [12].

An experimental programming environment has been developed: the Maiday system [9]. The goal is to provide an open-ended system to experiment with the method, the language and future software tools. To date, a structured editor and a translator to PASCAL have been designed and implemented. The user is helped in three ways: syntax when parsing occurs, context-sensitive constraints for type-checking and methodology, and strategy for the next action to be done. An interpreter is to be implemented. This basic tool will be used for experienced programmers to find out which strategies are involved in the planning stage of algorithms creation. Preliminary work has already been done [10] and will be continued with two cognitive psychologists.

Static objects, structured definitions and modularity are the three important features of the method. This method has some limitations. In particular, we are compelled to program 'in the small' because of the insufficiency of our concept of 'module'. The notion of data type has not been developed. Thus, the method is not adapted to solve problems whose data structures do not correspond to the structure of the results. One often encounters these problems when dealing with management analysis; they concern mostly what Jackson calls problems of 'structure clashes' [11]. In order to extend the deductive method of programming by systematizing problem solving dealing with sequences, we have decomposed the program construction into three steps:

- Choose the adequate intermediate data structures implied by the structure of the results;
- Define the result by using the previously chosen intermediate objects. An algorithm to define the result is developed with suitably chosen structures: data structures are not taken into account;
- Define the intermediate sequences by using the actual data of the problem.

Since the methodology leads to the introduction of useless intermediate objects, the resulting program is not very efficient. Our methodology concentrates on the elimination of these intermediate objects by successive program transformations.

The reader will find a development of those ideas in the companion paper: "Description and improvement of iterative program transformations" to be published in the next issue of this journal.

Acknowledgment

We would like to thank the referees and Peter King for their helpful comments.

References

[1] E.A. Aschcroft and W.W. Wadge, LUCID: a non procedural language with iteration, *Comm. ACM* **20** (1977) 519-526

[2] F. Bellegarde, J.P. Finance, B. Huc, J. Jarray, P. Lescanne, J. Marold, C. Pair, A. Quérè and J.L. Rémy, MEDEE: A type of language for the deductive programming method, RAIRO, Paris, (1979).
[3] F. Bellegarde, Rewriting systems on FP expressions that reduce the number of sequences they yield, *ACM Symposium on Lisp and Functional Programming*, Austin, TX (1984).
[4] M. Broy and M. Wirsing, Programming languages as abstract data types, *5éme Sym., Les Arbres en Algèbre et en Programmation*, Lille (1980).
[5] M. Broy and M. Wirsing, Algebraic definition of a functional programming language and its semantics models, *RAIRO Informat.* **17** (2) (1983).
[6] J.P. Finance, Static and computational semantics of a definitional language, in: E.J. Neuhold, Ed., *Formal Description of Programming Concepts* (North-Holland, Amsterdam, 1978).
[7] J.P. Finance and P. Lescanne, Trois approches sémantiques d'un langage permettant l'écriture des programmes analysés déductivement, *Actes du Congres AFCET Théorie et Technique de l'Informatique*, Tome 1 (Editions Hommes et Techniques, Paris, 1978).
[8] M.C. Gaudel, Génération et preuve de compilateurs basée sur une sémantique formelle des languages de programmation, Thèse d'état, INPL et INRIA, Nancy (1980).
[9] J. Guyard and J.P. Jacquot, Maiday: an environment for guided programming with a definitional language, *7th International Conference on Software Engineering*, Orlando, FL (1984).
[10] J.M. Hoc, Planning and direction of problem solving in structure programming: an empirical comparison between two method, *Internat. J. Man-Machine Studies* **15** (1981) 363-383.
[11] M.A. Jackson, *Principles of Program Design* (Academic Press, New York, 1975).
[12] E. Kolmayer, Evaluation de la méthode déductive de programmation, CRIN, Nancy, 79-R-074 (1979).
[13] C. Pair, Some theoretical aspects of program construction, *Internat. Summer School on Program Construction*, Munich (1978).
[14] C. Pair, Types abstraits et sémantique algèbrique des langages de programmation, CRIN, Nancy, 80-R-011 (1980).
[15] C. Pair, Sur les modèles des types abstraits algébriques, CRIN, Nancy, 80-P-052 (1980).
[16] Reddy, Programming with sequences, *ACM Southeast Regional Conference*, Knoxville, TN (1982).
[17] J. Souquières, Construction et transformations d'algorithmes itératifs, Thèse de docteur ingénieur, CRIN, Nancy (1982).
[18] J. Souquières and J.P. Finance, Description and improvement of iterative program transformations, *Sci. Comput. Programming*, to appear.